COMPLETE GROUND SOFTWARE RE-USE: THE COMMON GROUND APPROACH TO A RE-USABLE, SHARED GROUND SYSTEM

Priscilla L. McKerracher[1]
David S. Tillman
R. Michael Furrow
Leeha R. Herrera

[1]The Johns Hopkins University Applied Physics Laboratory (JHU/APL), Laurel, MD 20723-6099
Email: Priscilla.Mckerracher@jhuapl.edu

ABSTRACT

Institutions with existing spacecraft control systems often plan to "re-use" the architecture of the existing ground system on future missions. One means of re-using the existing system is to "snapshot" the existing software requirements, design, and code; deliver this package to the new project; and assign the new project team the task of evolving the new ground software from the snapshot.

Another approach to re-use is to establish a common team and employ a shared software repository. In this approach mission-specific requirements must be isolated from common requirements. This *common ground* approach has the potential for higher cost savings and improved overall product quality. Key challenges to this approach include the development of organizational, infrastructure, and technical solutions that support this model.

1. INTRODUCTION

The JHU/APL Space Department has a history of supporting Mission Operations for NASA spacecraft. Since the mid 1990s we have deployed a ground system with an architecture based on the use of Integral Systems EPOCH T&C®[9] product, which supports the core command, control and telemetry display features that are required. We "re-used" the initial EPOCH-based ground system, which was developed for the NEAR (Near Earth Asteroid Rendezvous) satellite, in the TIMED (Thermosphere, Ionosphere, Mesosphere Energetics and Dynamics), CONTOUR (COmet Nucleus TOUR), MESSENGER (MErcury Surface, Space ENvironment, GEochemistry, and Ranging), STEREO (Solar Terrestrial Relations Observatory), and New Horizons programs. All of these missions are NASA supported missions.

2. OLD SNAPSHOT PARADIGM FOR RE-USE

The original method employed for re-use of JHU/APL ground systems was to make a copy or "snap-shot" of the latest version of the older ground system, and to deliver the supporting documentation and code as a start for the next system. The new project then created a new "branch" from the existing heritage system. The assumption was that the requirements for the new system were similar to those for the heritage system; therefore, this approach was considered less costly than building a new ground system from scratch. In this approach there was re-use of the existing architecture, along with requirements and design. The TIMED and CONTOUR ground systems followed this model.

2.1 Characteristics

Although the snapshot approach is considered "faster" than starting from scratch; the savings are limited. Because the heritage system was not designed for re-use, the snap-shot has no guaranteed maturity, and the original project did not require the developer to avoid embedding mission-specific information into source code, the migration from original project to new project repository is not cost-free. That is, the amount of re-use savings with this approach is typically not as high as expected. Savings of 10-20% of development costs are typical; the project may hope for savings as high as 50-70%. The new developer must review the original documentation and code and locate the mission-specific parameters. Since there is a large body of documentation and code, this is not a trivial task. The developer must also evaluate the existing requirements against new project requirements and update the design and code accordingly. All modified documents must be reviewed and all modified code, must be re-tested and re-deployed in the new project environment.

With the snapshot approach there are no savings in the maintenance phase of the development. Any errors that are found and fixed in the new project must be separately communicated and manually updated in the heritage project. Separate Software Problem Reports (SPRs) must be entered, fixed and tracked in each project's problem tracking system. Often different developers are supporting different projects, and communication of problems may not occur as easily or as often as is desirable.

Finally, there are no strong incentives to minimize the divergence among the various conceptual and detailed design implementations. This results in "re-inventing the ground software wheel," which wastes limited project resources, as well as creative effort.

3. NEW COMMON GROUND PARADIGM

3.1 Goals

Despite the issues related with the "snapshot" re-use model, this model was adequate for the serial development of new ground systems on TIMED and CONTOUR. With the coming launch of three missions with overlapping development schedules (MESSENGER, STEREO and New Horizons) the ground systems applications group realized stronger drivers and opportunities for development of a new re-use model. The drivers included the goals to improve overall quality, to reduce Ground System software development costs, and to improve effort estimates and schedule compliance.

To achieve these goals we established a shared repository within a single "Common Ground" configuration management system. We required each of the three missions to contribute some initial development time to the start-up effort needed to restructure the existing requirements, design, and code. All identifiable mission-specific items were encapsulated and separated from the common codebase. In addition, updates were made to support an operating system upgrade that was incompatible with a commercial software library that was extensively used in the heritage code. A team of approximately six developers worked for ~six months to restructure and re-deploy the existing software. In addition they updated requirements and design documentation into standard formats, and held our required process reviews. After this initial development effort we deployed the first build of released software for the MESSENGER and STEREO projects. The first build of the New Horizons software was not required at this time. It was later configured and deployed at a cost of ~two staff-months of development time. This reflects a significant cost-savings for deployment of future first builds.

3.2 Fundamentals

In order to have significant software re-use and to realize the associated cost-savings, there are some required fundamentals. To begin with the top-level system requirements and design must remain applicable from mission to mission. Divergence in detailed requirements and design must be dramatically reduced from mission to mission. This approach implies some constraints on the "system" of flight software, flight hardware, ground software, as well as the concepts of operations employed by the Mission Operations and Integration and Test teams. The challenge is to find the right balance between constraints and flexibility such that the resulting system still meets the requirements of its user communities.

The mechanics of requirements development, capture, and configuration management must make the existing state of the ground system easily visible to the leads assigned to new missions. Visibility into the capabilities that are readily available and the operational concepts supported is a must. Each of these areas must support clear and flexible delineation of any mission-specific elements from the common elements.

To support a shared development environment, a single system is needed for software source code Configuration Management (CM) and for SPRs or Change Requests (CR's), for all missions. The CM system should favor capturing and presenting common information, but provide the flexibility to address mission-specific elements.

In addition, all personnel must develop a multi-mission mindset. Visibility into and responsibility for key functional areas across missions is required. This approach is a significant departure from the mission-oriented development teams of the past. The team must recognize the advantages of retaining corporate knowledge in functional areas, while providing natural disincentives to invent multiple solutions to the same problem. "Re-inventing the ground software wheel" is no longer supported.

3.3 Approach

From a software perspective, re-use can occur at the source file, library, application, or system level. At any of these levels, re-use of one or more of the following products is possible:

Concepts	Design
Requirements	Code
Architecture	Test Plans
Interface	User Guides

In order to achieve maximum re-use at several levels, we developed group organizational solutions, software infrastructure solutions, and implementation-specific technical solutions.

4. ORGANIZATIONAL SOLUTIONS

4.1 Product Lines

One of the most important elements of our approach is a reorganization along major functional areas or Product Lines (PL). The advantages of a PL organization include its ability to leverage knowledge in functional areas across multiple missions, and its tendency to naturally discourage deviations in approach from mission to mission.

With the snapshot approach we had multiple independent teams redeveloping the same functionality. Individuals on each team had to acquire a detailed understanding of essentially the same applications. This was an expensive and risky process since the skill level and the ability to infer how a pre-existing application operated varied among team individuals. Misunderstandings on how to

adapt subtle or undocumented design methodologies led to unexpected consequences. The intensity of the delivery schedule and the primary focus on the current mission led developers to take the most obvious path when modifying the software to support the mission. There was little incentive for them to generalize the software. There was also little opportunity to work with flight software developers and the Mission Operations team to adopt identical approaches across missions.

Our solution was to align *across* missions and *along* functional areas. For the "Common Ground" approach, we analyzed the existing Ground System applications and architecture and found that from a functional standpoint, the applications could be logically grouped into five Computer Software Configuration Items (CSCIs). We then established each logical grouping as a PL with a lead engineer (Product Line Lead [PLL]) responsible for the system engineering and architectural decisions within a PL. A System Engineer was appointed to oversee the system as a whole and to coordinate decisions and approaches used by the individual PLLs. The PL (and CSCI) areas are *Commanding*, *Telemetry*, *Planning*, *Assessment*, and *Tools*. Each CSCI is comprised of a set of underlying CSCs. The CSCs are primarily C/C++ programs running in a Unix environment and communicating with each other via Remote Procedure Calls (RPC) and Transmission Control Protocol/Internet Protocol (TCP/IP) socket mechanisms.

The Commanding PL is comprised of a number of JHU/APL-developed applications (or CSCs) associated with the translation of command mnemonics and arguments into binary representations. This includes CSC's which perform the Consultative Committee for Space Data Systems (CCSDS)[1,2,3,4,5] packetization of the commands, the framing of the packets, and the conversion to Command Link Transfer Units (CLTUs) required for radiation to the spacecraft.

The Telemetry PL encompasses CSCs associated with the acquisition of raw telemetry (e.g., from the selected antenna interface). There are processes required for extracting meta-data from the Telemetry Transfer Frame wrappers, and reconstructing and distributing packets to the commercial-off-the-shelf (COTS) real-time telemetry component.

The Planning PL includes the applications that create scripts and binary data needed for loading parameters, structures, and onboard executables to the spacecraft. This PL is also responsible for interpreting the results of downlinks (or dumps) of those objects for the purpose of verifying correct transmission to the spacecraft. For missions that make extensive use of onboard programming (i.e., macros), it provides the post-processing required to manage the allocation of commands to macros, the creation of time-tags, and the coordination of the load. Software-based spacecraft simulators used for command verification also fall within this PL.

Assessment CSCs are responsible for archiving and retrieving the telemetry needed for analysis or for spacecraft health and assessment purposes. Multiple CSCs work in an offline capacity to process the quantities of data delivered post-pass. This CSC supports offline processing of alarms in housekeeping data collected between passes and saved to the recorder. It also supports processes that perform short term and long-term analyses of selected points to support the detection of unsafe trends.

The Tools CSCI is a collection of applications, whose primary unifying characteristic is that they supply some unique capability that would not be economical to duplicate in house. Many of these pieces are COTS or government-off-the-shelf (GOTS) applications, occasionally augmented by JHU/APL-developed components.

4.2 Common Ground Team

All of the PLLs are directly involved with development as well as with the direction of a small pool of developers (~ 10). Development activity is coordinated across missions so that the developers can develop common functionality for all missions at the same time. The PLL is therefore motivated to minimize the differences between missions. If the ground software differences stem from differences in flight software or Mission Operation approaches, the PLL will bring this to the attention of the relevant parties and lobby for a common approach, if possible. The PLL and developers are well versed in the detailed requirements for their functional area for all missions and thus are in a position to recommend a common approach that the other team members may not have considered.

4.3 Working Instructions

With the introduction of any new process, it is essential to educate and support the team in adapting to the new methods. We needed a process that would allow us to deliver major, minor, and patch releases with minimal overhead, but assured integrity. The CM tool's facilities had to be employed in such a way as to support the concept of a single code base with mission-specific specializations and to do so in a manner straightforward enough to not impede the developers.

A set of procedures, developed and maintained online, provide detailed instructions for supporting all the possible development scenarios (normal development, patch release, etc.). Additional procedures detail the steps Build Managers and the Configuration Managers execute in producing deliveries.

5. INFRASTRUCTURE SOLUTIONS

The Common Ground goal of maximal re-use is supported by a number of commercial tools and an existing architecture.

5.1 DOORS®, for Requirements Re-use

The DOORS® tool from Telelogic provides a powerful and flexible repository for maintaining requirements, test plans, and other documents under CM. Tagging common requirements in DOORS®, allows us to present a coherent picture of the native capability of the Ground System software. This forms an excellent starting point for cost estimation in the proposal phase and for choosing the path best supported during the initial spacecraft system design phase. By starting with a mature set of requirements, and an associated concept of operations, we have significant leverage at the flight software and Mission Operations levels to steer design decisions in a direction already supported.

5.2 Re-use of Architecture

The Common Ground real-time architecture is based on the TIMED Ground System. The key architectural element is the use of configuration files to designate a number of processes which "live" and "die" together and which communicate via defined internal interfaces over TCP/IP socket connections. This collection of configurable, dependent processes is known as a "stream" in EPOCH T&C.

5.3 Re-use of CSC and Design Documents

To facilitate document visibility, an electronic folder on a shared server provides the focal point for detailed design documentation and review material. Subfolders exist for each PLL. Within the PLL folders are individual CSCs folders. At the top level is a "Reviews" folder that contains a subfolder for every Requirements and Design review held for any component. The most recent detailed design document is actively maintained under its CSCI folder. Mission-specific details are placed in mission-specific appendices as required. A standard naming convention and central access greatly enhances the visibility to all developers, testers, and interested parties on the details of ground software functionality and design. This posting method also provides a model for design presentation for new and less experienced developers.

5.4 Re-use With CM Synergy

The CM Synergy system serves as the repository for all the JHU/APL-developed software in the Ground System as well as for selected GOTS products. The system supports the concepts of projects, directories, and objects. Each CSC application is represented as a project. Each project has a top level directory, named for the CSC. Within that directory are typically two branches. The *app-specific* branch holds the directories containing objects that are not used by any other application. Typically there is a *src* subdirectory. There might also be a *test* or a *config* subdirectory.

The other branch consists of a directory called *Common*. It is in this directory that all files shared across more than one application are listed. Distinguishing between application specific and common files in this way provides a strong visual message to the developer – changes to files in the Common area *must* be coordinated with the PLL to ensure there are no detrimental effects on other applications.

Within the *src* directory, each application is required to have a *Makefile* that conforms to certain rules. In particular, it is responsible not only for building the associated executable, but it also manages the deployment of the executable, scripts, and any required configuration files to the correct relative directory when the "release" target is invoked. By setting up the local (to the development workstation) deployment directory structure to mimic the production structure, the developer is able to control the placement of files critical to their application. This greatly simplifies the deployment process as the most knowledgeable person is capturing the specific deployment requirements in the *Makefile*.

Aggregate projects are defined at the PL (or CSCI) level, which contain the CSC projects that make up the CSCI. Additionally, these CSCI-level projects contain a Makefile whose sole role is to propagate commands to build the system from the system-level Makefile to the CSC Makefiles. Again, by doing this and having the PL maintain this Makefile, we achieve the goal of having the person closest to the change making the change. The CSCI-level Makefile only changes when a new CSC (and consequently project) is added to the CSCI. The system-level Makefile does not need to be modified.

The top level aggregate project is called *ground* and includes as subprojects all of the CSCI aggregate projects plus pseudo-CSCI aggregate projects that manage the Build scripts, system-wide configuration files, support functions, and utilities that are needed to complete the system.

Figure 1 presents the Common Ground CM concept and presents the CM Synergy capability to associate a "platform" attribute with any object to tag files and directories as mission-specific. By default, directories and files are *generic.* If an application needs to have mission-specific parameters, then multiple *versions* of the file containing the mission-specific elements are maintained. This concept is illustrated in Figure 2. Note that these files have the same name and are all associated with the project. Each version has a different value for the "platform" attribute.

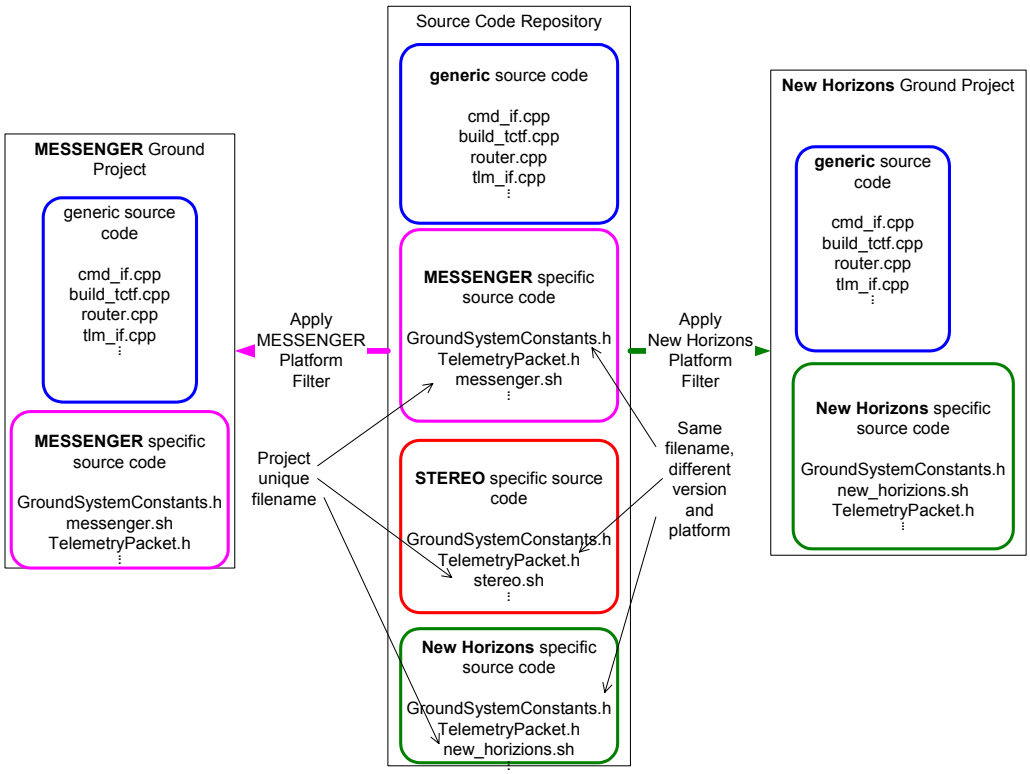

Figure 1. Common Ground Configuration Management Concept

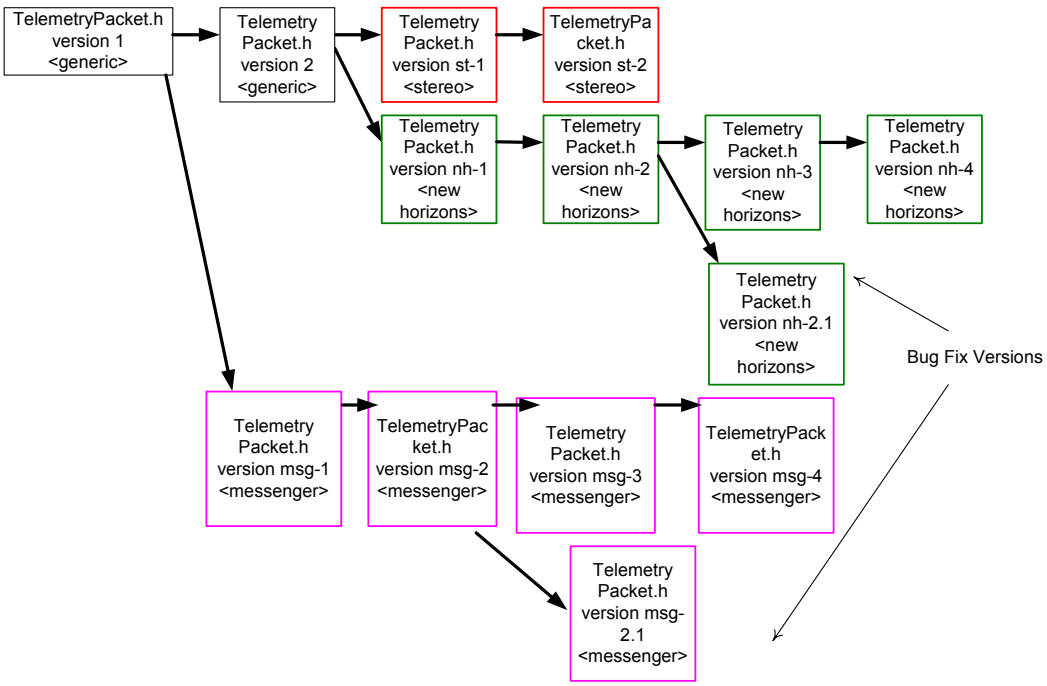

Figure 2. Configuration Management Support for Version Branching

When the time comes to build the application, the developer specifies the mission and CM Synergy creates a file set corresponding to that mission. It will favor mission-specific versions of files if they exist; otherwise it will supply the generic version. It will also support branching to support bug fixes to prior releases which require maintenance fixes.

In some cases, differences between applications from mission to mission are great enough that the actual file complement differs between the missions. That is, the application will require different files for the different missions, rather than just different versions of the same file. CM Synergy manages this through the management of the directory object that contains the files. When the contents of that directory must be different between missions, the developer creates mission-specific *versions* of that directory, each with the desired file complement.

In all cases, visual information provided by CM Synergy makes clear which files and directories are generic and which are mission-specific, and it is easy for the developer to quickly identify what the differences are. In some cases we are able to redesign the application to eliminate the need for mission-specific versions of files. CM Synergy readily supports the merging of files and the establishment of a single generic copy used by all applications.

5.5 Common Versions of COTS/GOTS

The EPOCH software is maintained by Integral Systems, and is delivered and built according to documented procedures on a general-purpose (not mission-related) workstation at JHU/APL. All missions use the same version of EPOCH, operating system, compiler, and commercial libraries. One of the key benefits of the Common Ground approach is that the effort associated with migrating applications to new compiler, operating system (OS), and commercial library versions is distributed across the missions. This, in addition to our adherence to use of ISO (International Organization for Standardization) standard language features and POSIX (Portable Operating System Interface based on uniX) system calls where available, simplifies the long-term maintenance of the system.

6. A SAMPLE TECHNICAL SOLUTION

Using an object-oriented design approach can further ease the concurrent development of multiple projects with similar, but not identical data structures or functionality. Inevitably, because the projects are different, functionality will vary and code for common tasks will need to be specialized in different areas. How should these situations be handled? With the support of proper code management and build processes, the areas of code that need to differ can be handled by using a parallel class pattern that employs parallel versions of the affected file or class.

Consider the example where two projects each read data from a socket, sort the data, and store the data in merged and sorted files as illustrated in Figure 3. In general the projects are similar, except that the format of the data to be stored will differ.

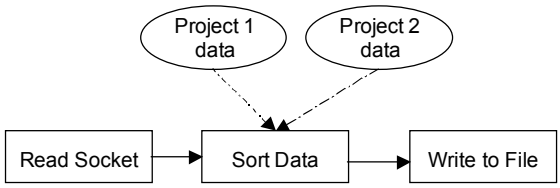

Figure 3. Example of Sort Required for Two Projects with Different Data Formats

Only the code dealing with the interpretation of data will need to be customized for each project. An object-oriented solution which employs mostly common classes is possible. Project specific modifications can be limited to a few select classes. In this example, two common classes are created first. A Sort Data class is created which uses a second Data Definition class. The Data Definition class contains the basic structure definition information needed to read the data. The common Data Definition class can be replaced by a project-specific version. The Common Sort Data class is unaffected by the particular version of the structure class used, and therefore remains a “generic” or “common” class.

Use Parallel Classes pattern when:

- Multiple projects are being developed by the same team and have a high degree of code sharing.
- A common architecture has been developed and will be reused for specific projects.
- Resources are limited and need to be shared across multiple projects.

6.1 Structure

The class structure to this pattern is fairly simple. This pattern relies on a well-structured object oriented architecture, which separates the functionality of a project appropriately and delineates areas adaptable to common code. In the data sorting example, two common classes are defined: a Sorter class and a Data Definition class. The common Sorter Object uses the common Data Definition class, thus establishing a user dependency relationship between the classes, as shown by the dotted arrow in Figure 4.

All projects begin with the Common code base. As the project develops and specific project needs are identified, the class structure may be branched by replacing the

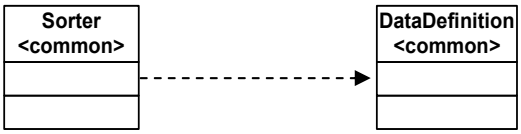

Figure 4. Common Sorter Class Uses Common Data-Definition Class

common Data Definition class with a parallel, project-specific Data Definition class as shown in Figure 5. It is important to note that the class names and file names are the same. This is where an advanced CM tool allows objects of the same name to be managed through the use of a CM object field that maintains the common or project specific identifier, keeping the parallel objects unique.

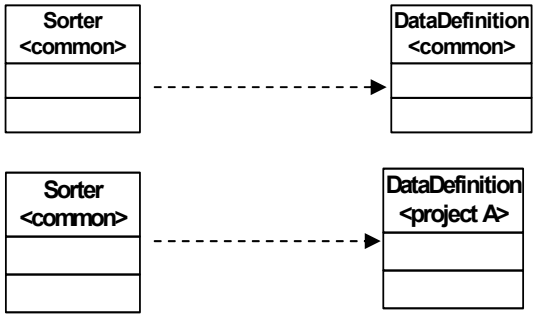

Figure 5. Common Sorter Class Can Use Either Project Data Definition Class

If it is obvious the branching will be necessary, it is a good idea to setup a class hierarchy to reduce the amount of redundant code for the different projects. The nature of the project may itself lend to a class hierarchy with a common parent class to establish the interface and project specific child classes to define the details of the interface as shone in Figure 6.

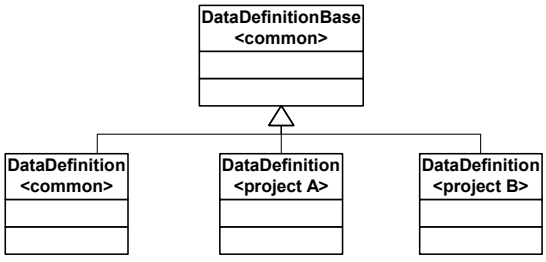

Figure 6. DataDefinition Base Class provides a Generalization of DataDefinition Classes for Each Project

Since changes to DataDefinition <project A> do not affect DataDefinition <project B>, with the parallel class pattern each project can manage the details of their interface independently.

6.2 Consequences

Parallel Class Structures:
Promotes Reuse and Code Share. A common code base can be easily used by multiple projects while still allowing the flexibility to customize necessary areas of code.

Requires Configuration Management of Classes. Requires extra CM control to keep track of parallel versions and associations between Computer Software Components (CSCs).

Requires Change Control over Common Classes. Since Common Classes may be used in multiple projects, it is necessary to have a process implemented to submit requests to change Common Classes, to ensure that it does not have adverse affects on other code.

Developers will have to adjust to the Structure. It may cause confusion at first for developers when modifying code. The purpose of the Parallel Structure should be well explained as well as the general guidelines in modifying common classes, creating branches, and maintaining the makefiles for each CSC.

6.3 Implementation

The following are a few things to consider when implementing the Parallel Class pattern:

Creation of Common Code Base. It may be difficult to identify what is common when trying to create a code base for future projects. One solution is to start with an existing project's code base, which could be modified to support one of more of the next generation projects. The first step would be to make sure the existing code is in CM and to then identify the areas that can be common across the new missions. The next step is to go back though the new code base and remove any project specific attributes by either trying to use initialization files or header files to store necessary hard code values or by assuming class branching will be necessary and implementing a rudimentary class as a Base class and using inheritance to reduce to work for each project specific class.

Creating Parallel Classes. Only branch a class when project specific modifications are needed. Each branch requires work in maintaining similar code and algorithms. Try to implement solutions that reduce the need of Parallel Classes when possible. Each time a project specific version is created, it will be necessary to make the file association modifications within the CM tool for the project. Depending on the file structure used, it may be necessary to modify the location of the class within the makefile. Once a branch is created, it is best if all dependant CSCs are modified to use the project specific version.

6.4 Configuration Management

The CM can potentially be the most complex aspect of this pattern, depending on the CM tool that is used.

For a simpler tool, it will probably be necessary to designate a directory structure to handle common and project specific classes. At a high level there should be a designated common or generic directory and a directory for each project. Under those directories would be the necessary structure, which would be the same for each branch. The project specific directory branches may be fairly empty compared to the common directory, if most of the code used is common. It is necessary for developers to be aware of which branch is being used and communication is necessary between the developers so that all may be notified when a branch has been made. Under this type of system it is necessary to ensure the makefiles are maintained to pull in the correct files.

With a more sophisticated object based tool, some of the work is reduced. It is easier to maintain common and project specific versions in a tool that is not dependant on a directory structure but maintains everything as objects in a database. Rational's ClearCase and Telelogic's CM Synergy are examples of this type of tool. The tool will have the capability to map the branching for each object and removes the need to maintain a directory structure.

7. CONCLUSION

Once the main functionality is implemented for one mission and released, the specializations needed for the remaining missions are addressed. This approach results in substantial savings, as the cost of the initial implementations of significant functions get distributed across the active missions. Costs for adapting the functionality for the other missions range from significant (typically memory object management) to nothing at all, depending on the success of the lobbying efforts.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the heritage work provided by the software development teams for the NEAR and TIMED missions. We especially acknowledge the hard work and support of the entire "Common Ground" software development team, the dedicated support by system administration personnel and acceptance test team, as well as the patience, input and support from the user community we serve: flight subsystem developers and testers, integration and test team personnel, and of course, the mission operations teams on all of our missions.

9. REFERENCES

[1] *Telemetry Summary of Concept and Rationale*, CCSDS 100.0-G-1. Green Book. Issue 1. Washington, D.C.: CCSDS, December 1987. http://www.ccsds.org/documents/100x0g1.pdf

[2] *Packet Telemetry*, CCSDS 102.0-B-5. Blue Book. Issue 5. Washington, D.C.: CCSDS, November 2000. http://www.ccsds.org/documents/102x0b5.pdf

[3] *Packet Telemetry Service Specification*, CCSDS 103.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, June 2001. http://www.ccsds.org/documents/103x0b2.pdf

[4] *Telecommand Summary of Concept and Rationale*, CCSDS 200.0-G-6. Green Book. Issue 6. Washington, D.C.: CCSDS, January 1987. http://www.ccsds.org/documents/200x0g6.pdf

[5] *Telecommand Part 1 – Channel Service*, CCSDS 201.0-B-3. Blue Book. Issue 3. Washington, D.C.: CCSDS, June 2000. http://www.ccsds.org/documents/201x0b3.pdf

[6] *Telecommand Part 2 – Data Routing Service*, CCSDS 202.0-B-3. Blue Book. Issue 3. Washington, D.C.: CCSDS, June 2001. http://www.ccsds.org/documents/202x0b3.pdf

[7] *Telelogic DOORS®* http://www.telelogic.com/products/doorsers/index.cfm

[8] *Telelogic CM Synergy* http://www.telelogic.com/products/synergy/index.cfm

[9] *EPOCH T&C® (Telemetry and Command, Integral Systems* http://www.integ.com/EPOCHT&C.HTM